HOW TO SET UP A CONTINUOUS INTEGRATION & DELIVERY (CI/CD) PIPELINE

An effective continuous integration (CI) and continuous delivery (CD) pipeline is essential for modern DevOps teams to cope with the rapidly evolving technology landscape. Combined with agile concepts, a fine CI/CD pipeline can streamline the software development life cycle resulting in higher-quality software with faster delivery.

In this article, I will:

- Define a CI/CD pipeline
- Explain the four stages
- Share examples
- Show you how to set up a Jenkins pipeline
- And more!

(This article is part of our DevOps Guide. Use the right-hand menu to go deeper into individual practices and concepts.)

What is a CI/CD pipeline?

The primary goal of a CI/CD pipeline is to automate the software development lifecycle (SDLC).

The pipeline will cover many aspects of a software development process, from writing the code and running tests to delivery and deployment. Simply stated, a CI/CD pipeline integrates automation and continuous monitoring into the development lifecycle. This kind of pipeline, which encompasses all the stages of the software development life cycle and connects each stage, is collectively called a

CI/CD pipeline.

It will reduce manual tasks for the development team which in turn reduces the number of human errors while delivering fast results. All these contribute towards the increased productivity of the delivery team.

(Learn more about deployment pipelines and the role of CI/CD.)

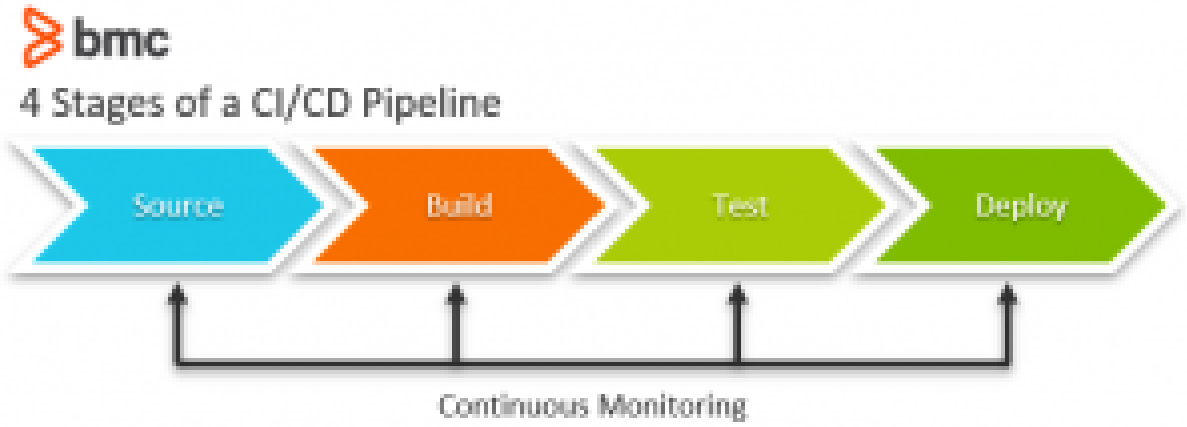
Stages in a CI/CD pipeline

A CI/CD pipeline can be divided into four main stages:

1. Source
2. Build
3. Test
4. Deployment

Each subsequent stage must be completed before continuing to the next stage. All the stages are continuously monitored for errors or any discrepancies, and feedback is provided to the delivery team.

In an agile context, each development, whether bug fix or feature improvement, falls into the CI/CD pipeline before deploying to production.



Source stage

This is the first stage of any CI/CD pipeline. In this stage, the CI/CD pipeline will get triggered by any change in the program or a preconfigured flag in the code repository (repo). This stage focuses on source control, covering version control and tracking changes.

What exactly happens in this stage? If the automated workflow detects a change in the central repository (commit, new version), it will trigger tasks such as code compilation and unit testing.

Common tools in the source stage include:

- GIT
- SVN
- Azure Repos
- AWS CodeCommit

Build stage

This second stage of the pipeline combines the source code with all its dependencies to an executable/runnable instance of the development. This stage covers:

- Software builds
- Other kinds of buildable objects, such as Docker containers

This stage is the most important one. Failure in a build here could indicate a fundamental issue in the underlying code.

Additionally, this stage also includes the build artifact handling. The storage process of building artifacts can be centralized using a centralized artifact repository like yarn, JFrog, or a cloud-based solution such as Azure Artifacts. This gives us the ability to roll back to the previous build if there are any issues with the current build.

Tools that support the build stage:

- Gradle
- Jenkins
- Travis CI
- Azure Pipelines
- AWS Code Build

Test stage

The test stage incorporates all the automated testing to validate the behavior of the software. The goal of this stage is to prevent software bugs from reaching end-users. Multiple types of testing from integration testing to functional testing can be incorporated into this stage. This stage will also expose any errors with the product.

Common test tools include:

- Selenium
- Appium
- Jest
- PHPUnit
- Puppeteer
- Playwright

Deploy stage

This is the final stage of the pipeline. After passing all the previous stages, the package is now ready to be deployed. In this stage, the package is deployed to proper environments as first to a staging environment for further quality assurance (QA) and then to a production environment. This stage can be adapted to support any kind of deployment strategy, including:

- Blue-green deployments
- Canary deployments
- In-place deployments

The deployment stage can include infrastructure provisioning, configuration, and containerization using technologies like Terraform, Puppet, Docker, and Kubernetes. Other tools include:

- Ansible
- Chef
- AWS Code Deploy
- Azure Pipelines - Deployment
- AWS Elastic Beanstalk

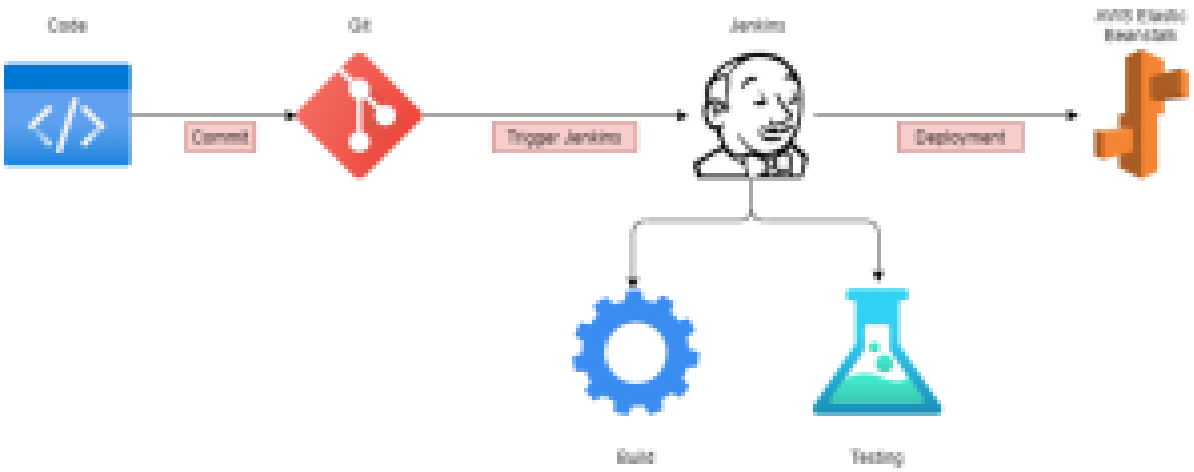
(Compare the software deployment & release processes.)

CI/CD pipeline examples

Now, let's look at two CI/CD pipelines in action. The first is a traditional pipeline, then we'll turn to a cloud-based pipeline.

Traditional CI/CD pipeline

The following pipeline represents a simple web application development process.



Traditional CI/CD pipeline

1. The developer develops the code and commits the changes to a centralized code repository.
2. When the repot detects a change, it triggers the Jenkins server.
3. Jenkins gets the new code and carries out the automated build and testing. If any issues are detected while building or testing, Jenkins automatically informs the development team via a preconfigured method, like email or Slack.
4. The final package is uploaded to AWS Elastic Beanstalk, an application orchestration service, for production deployment.
5. The elastic beanstalk manages the provisioning of infrastructure, load balancing, and scaling of the required resource type, such as EC2, RDS, or others.

This is a good example, but yours does not need to match it exactly. The tools, processes, and complexity of a CI/CD pipeline will depend on the development requirements and business needs of the organization. The result could be either:

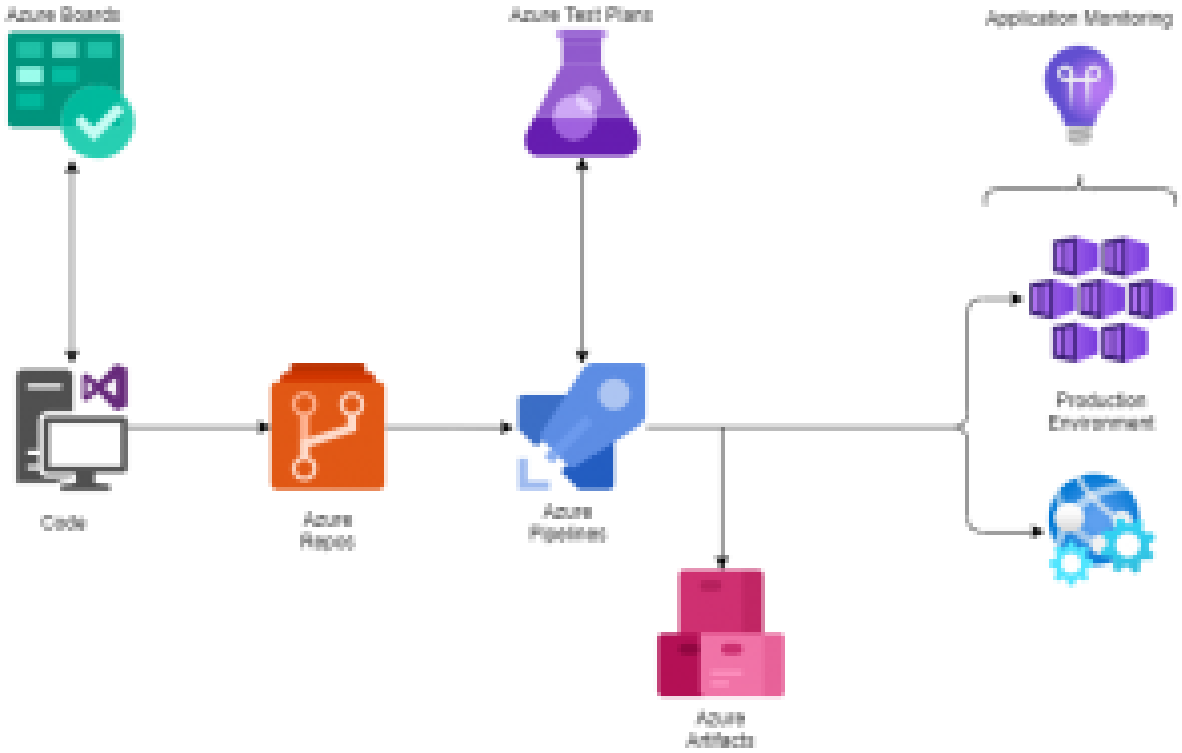
- A straightforward four-stage pipeline

- A multi-stage concurrent pipeline, including multiple builds, different test stages (smoke test, regression test, user acceptance testing), and a multi-channel deployment (web, mobile)

Cloud-based CI/CD pipeline

With the increased adoption of cloud technologies, the growing trend is to move the DevOps tasks to the cloud. Cloud service providers like Azure and AWS provide a full suite of services to manage all the required DevOps tasks using their respective platforms.

The following is a simple cloud-based DevOps CI/CD pipeline entirely based on Azure (Azure DevOps Services) tools.



Cloud-based CI/CD pipeline

1. A developer changes existing or creates new source code, then commits the changes to Azure Repos.
2. These repo changes trigger the Azure Pipeline.
3. With the combination of Azure Test Plans, Azure Pipelines builds and tests the new code changes. (This is the Continuous Integration process.)
4. Azure Pipelines then triggers the deployment of successfully tested and built artifacts to the required environments with the necessary dependencies and environmental variables. (This is the Continuous Deployment process.)
5. Artifacts are stored in the Azure Artifacts service, which acts as a universal repository.
6. Azure application monitoring services provide the developers with real-time insights into the deployed application, such as health reports and usage information.

In addition to the CI/CD pipeline, Azure also enables managing the SDLC using Azure Boards as an agile planning tool. Here, you'll have two options:

- Manually configure a complete CI/CD pipeline
- Choose a SaaS solution like Azure DevOps or DevOps Tooling by AWS

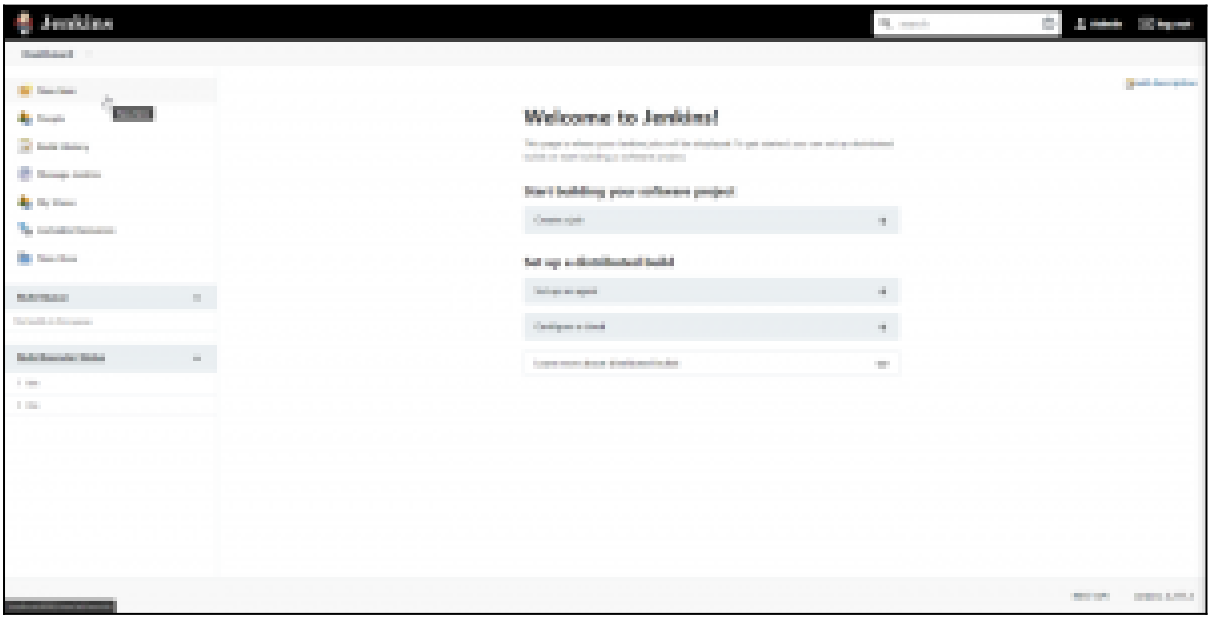
How to build a CI/CD pipeline using Jenkins

In this section, I'll show how to configure a simple CI/CD pipeline using Jenkins.

Before you start, make sure Jenkins is properly configured with the required dependencies. You'll also want a basic understanding of Jenkins concepts. In this example, Jenkins is configured in a Windows environment.

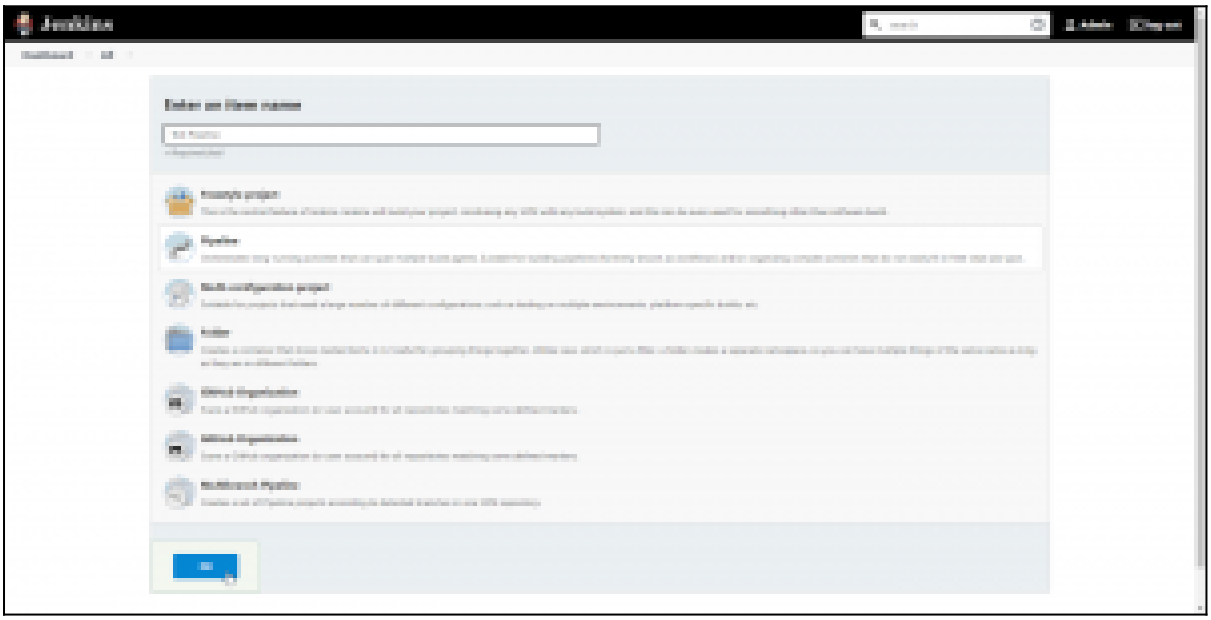
Step 1: Opening Jenkins

Login to Jenkins and click on "New Item."



Step 2: Naming the pipeline

Select the "Pipeline" option from the menu, provide a name for the pipeline, and click "OK."



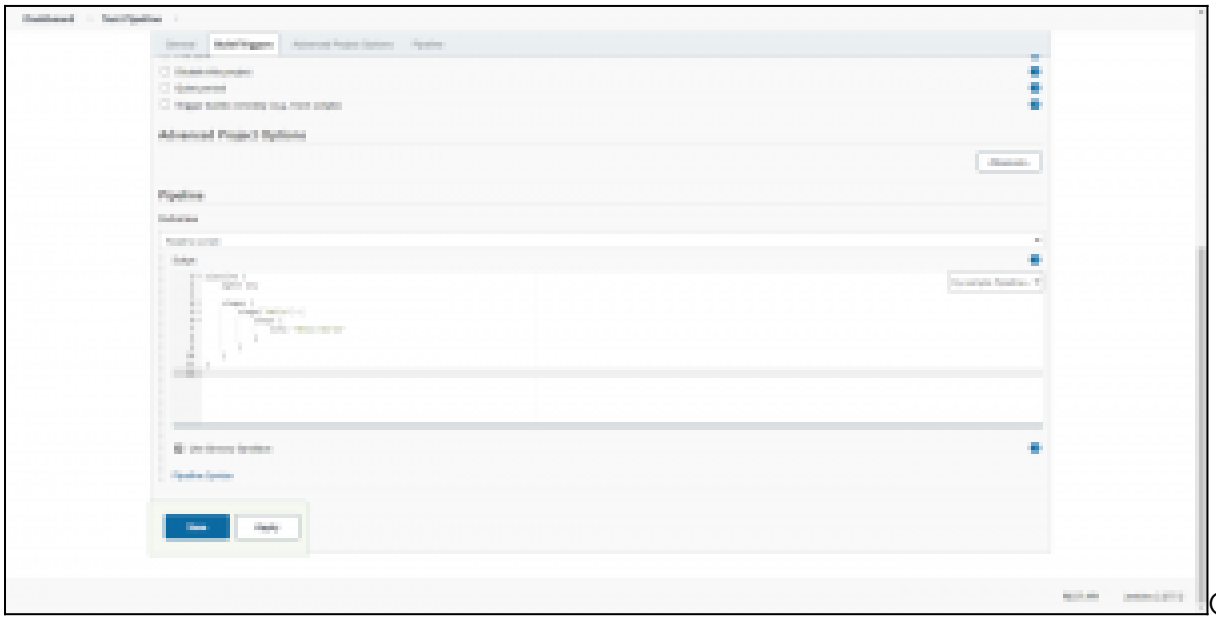
Step 3: Configuring the pipeline

We can configure the pipeline in the pipeline configuration screen. There, we can set build triggers and other options for the pipeline. The most important section is the "Pipeline Definition" section, where you can define the stages of the pipeline. Pipeline supports both declarative and scripted syntaxes.

(Refer to the official Jenkins documentation for more detail.)

Let's use the sample "Hello World" pipeline script:

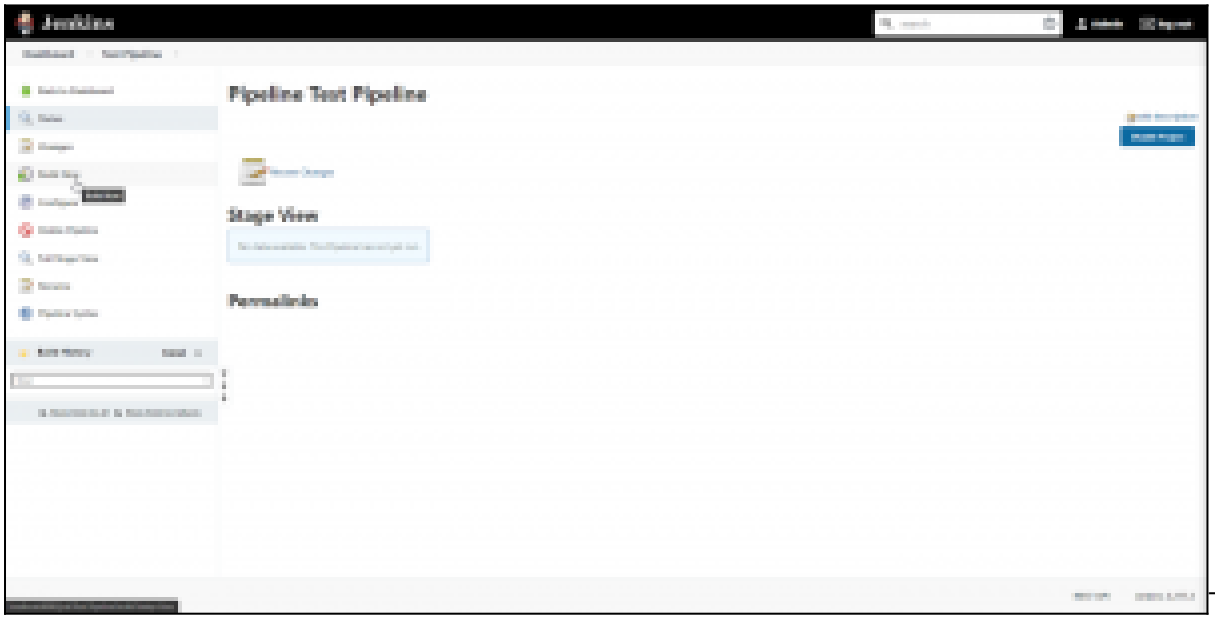
```
pipeline {
    agent any

    stages {
        stage('Hello') {
            steps {
                echo 'Hello World'
            }
        }
    }
}
```



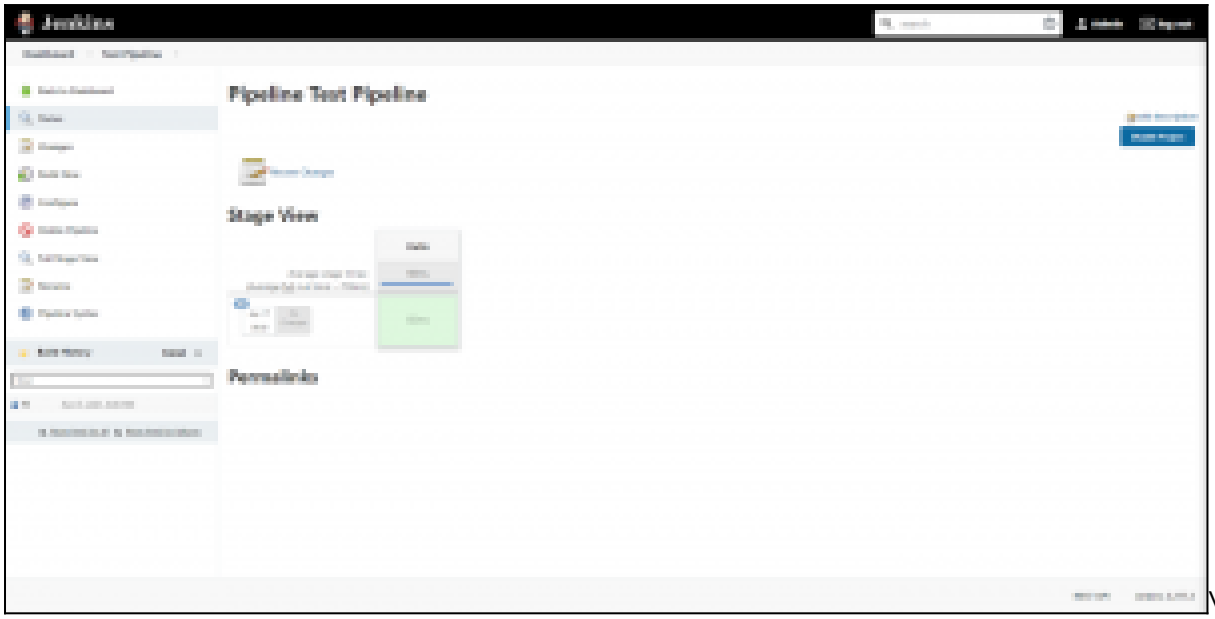
Click on Apply and Save. You have configured a simple pipeline!

Step 4: Executing the pipeline

Click on "Build Now" to execute the pipeline.



This will result in the pipeline stages getting executed and the result getting displayed in the "Stage View" section. We've only configured a single pipeline stage, as indicated here:



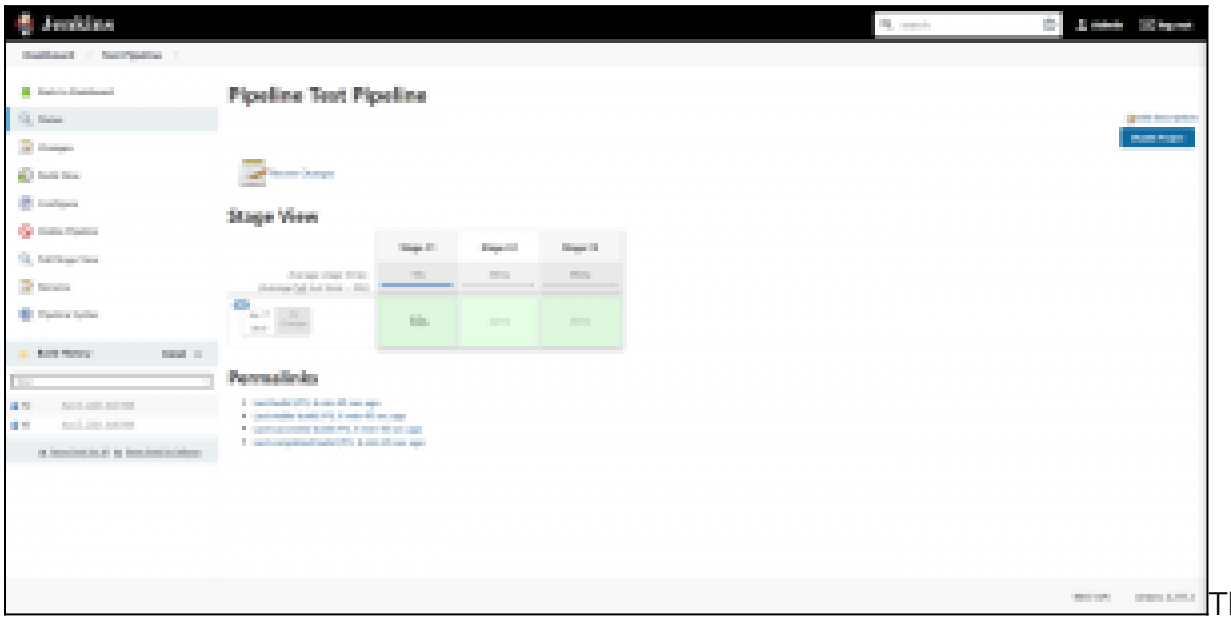
We can verify that the pipeline has been successfully executed by checking the console output for the build process.

Step 5: Expanding the pipeline definition

Let's expand the pipeline by adding two more stages to the pipeline. For that, click on the "Configure" option and change the pipeline definition according to the following code block.

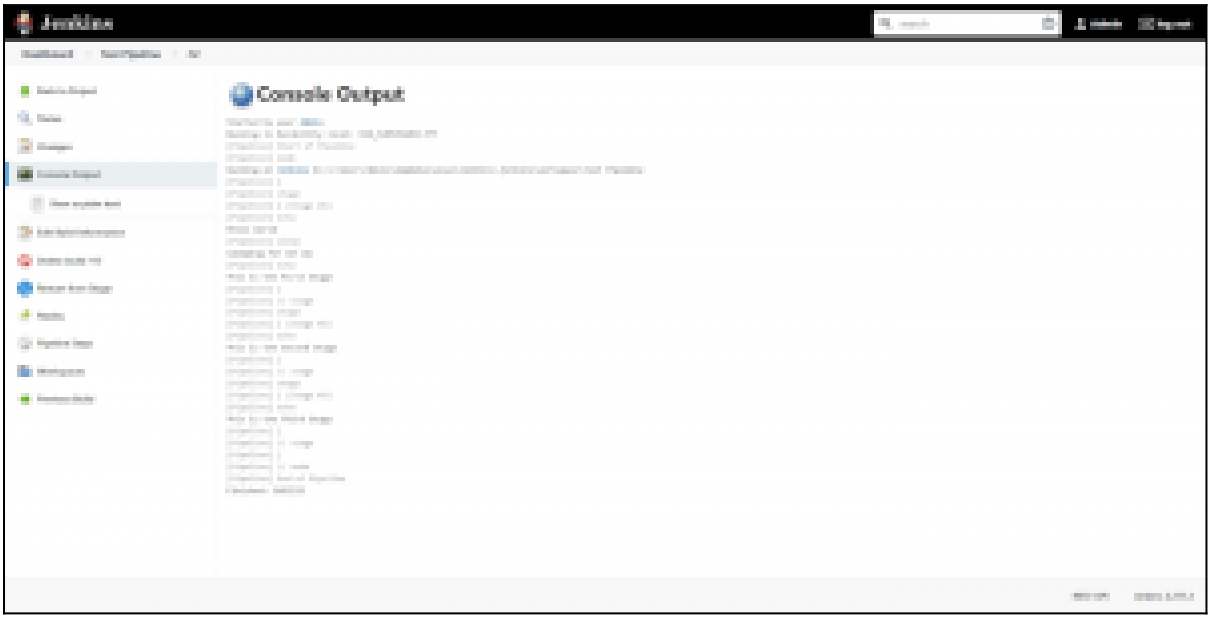
```
pipeline {
    agent any

    stages {
        stage('Stage #1') {
            steps {
                echo 'Hello World'
                sleep 10
                echo 'This is the First Stage'
            }
        }
        stage('Stage #2') {
            steps {
                echo 'This is the Second Stage'
            }
        }
        stage('Stage #3') {
            steps {
                echo 'This is the Third Stage'
            }
        }
    }
}
```

Save the changes and click on “Build Now” to execute the new pipeline. After successful execution, we can see each new stage in the Stage view.



The following

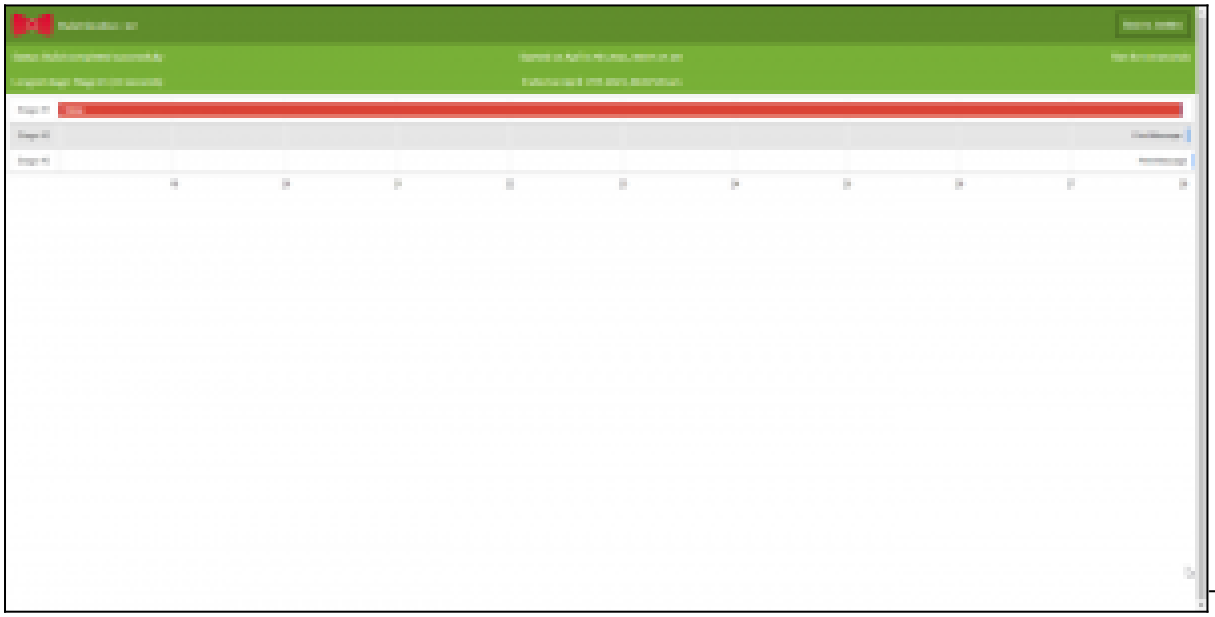
console logs verify that the code was executed as expected:



Step 6: Visualizing the pipeline

We can use the "Pipeline timeline" plugin for better visualization of pipeline stages. Simply install the plugin, and inside the build stage, you will find an option called "Build timeline."

Click on that option, and you will be presented with a timeline of the pipeline events, as shown below.

 That's it! You've successfully configured a CI/CD pipeline in Jenkins.

The next step is to expand the pipeline by integrating:

- External code repositories
- Test frameworks
- Deployment strategies

Good luck!

CI/CD pipelines minimize manual work

A properly configured pipeline will increase the productivity of the delivery team by reducing the manual workload and eliminating most manual errors while increasing the overall product quality. This will ultimately lead to a faster and more agile development life cycle that benefits end-users, developers, and the business as a whole.

Learn from the choices Humana made when selecting a modern mainframe development environment for editing and debugging code to improve their velocity, quality and efficiency.

Related reading

- BMC DevOps Blog
- SRE vs DevOps: What's The Difference?
- How Containers Fit in a DevOps Delivery Pipeline
- How & Why To Become a Software Factory
- Book Review of The Phoenix Project: DevOps For Everyone
- Enterprise DevOps: Increase Agility Across the Enterprise